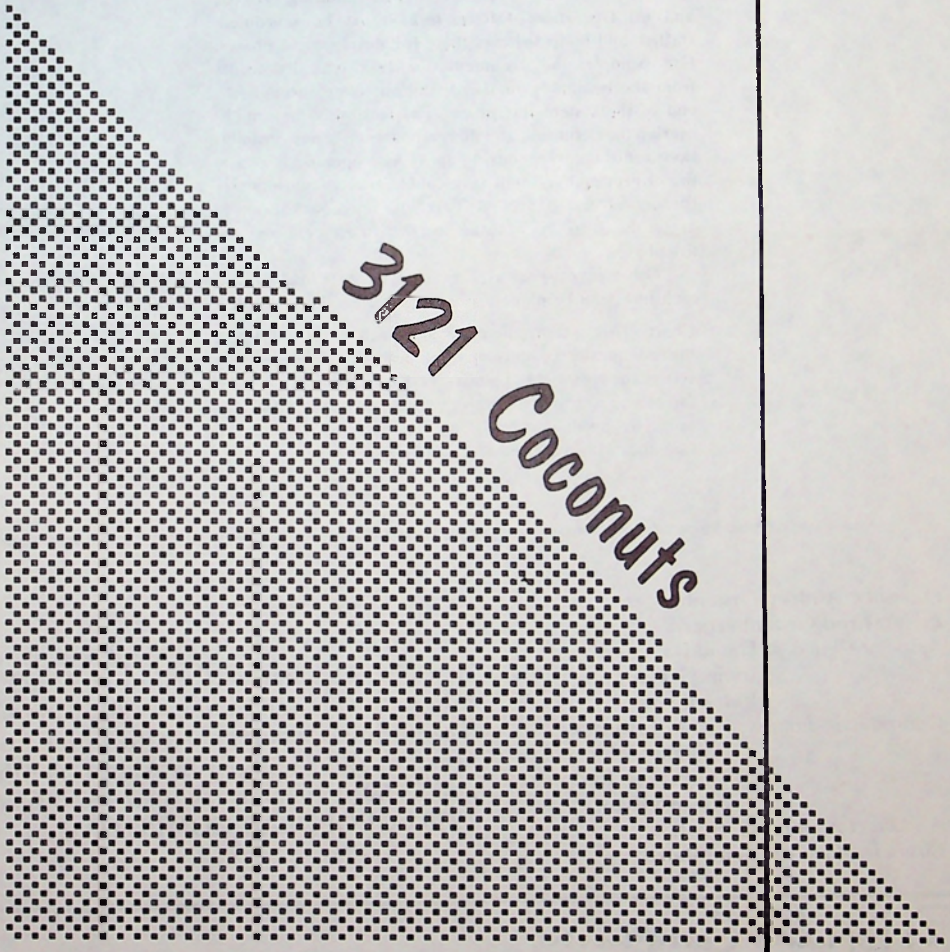
Popular Computing

The world's only magazine devoted to the art of computing.

Volume 8 Number ~~10~~ 9

September 1980

90



BOOK REVIEW

Application Design Handbook For Distributed Systems

Robert L. Patrick.
CBI Publishing Company, Inc., 51 Sleeper Street, Boston, Massachusetts 02210. 1980. 288 pages, $19.95 hardbound

This is a splendid book. It is a handbook for the design of application programs for a small computer environment. It is *not* a primer and it does *not* teach applications design per se. Writing principally for the systems analyst who is experienced with large host systems, the author highlights differences between the traditional centralized batch and on-line distributed system environments. He stresses how design is an iterative process and why the designer must be a visionary to plan for changing requirements.

Chapter 1 gives an overview of five major *phases* in developing an application for a small machine environment: (1) Definition, (2) Design, (3) Programming, (4) Test, and (5) Operation. Fifteen *tasks* must be scheduled, staffed, and budgeted over these five development phases. For example, the documentation task typically occurs from the beginning of the definition phase through the end of the system test phase. The feasibility task might overlap the definition and design phases. If a post-installation audit occurred during the final (operation) phase, the project management task could proceed continuously through all five phases. A Task Phasing Chart serves to depict the collection of tasks and their placement over the five phases.

Other managerial aspects, such as staff loading and cash flow, can be plotted in parallel on the Task Phasing Chart. This system life-cycle approach helps to expose the complexity of development activities. The author's systematic treatment of phases, tasks, activities, and charts forces you to think through the problems likely to be obtained in producing a system that performs the desired functions within a schedule and a budget.

Continued on page 15

Publisher: Audrey Gruenberger
Editor: Fred Gruenberger
Associate Editors: David Babcock
Irwin Greenwald
Patrick Hall
Contributing Editors: Richard Andree
William C. McGee
Thomas R. Parkin
Edward Ryan
Art Director: John G. Scott
Business Manager: Ben Moore

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year. For all other countries, add $3.50 per year. Back issues $2.50 each. Copyright 1980 by POPULAR COMPUTING.

The Coconuts Problem

The magazine Small Systems World presents a problem each month. In the May issue, Taskmaster Challenge number 37 is given here. The challenge calls for writing a program (presumably in some language that will run on one of IBM's small business computers). The problem here is the old coconuts problem, as submitted by Contributing Editor Judy Lawlor. Judy, bless her heart, has managed to distort a fine old traditional problem; the new problem (which has infinitely many answers) is mis-stated; and, as we shall see, it is not a computing problem at all.

The Challenge Master and originator of this month's task is Judy Lawler of Phoenix, Arizona. Judy is a contributing editor to Small Systems World.

Shipwrecked in a hurricane, five sailors in a rowboat landed on a small island inhabited by one monkey and numerous coconut trees. The trees had just finished bearing a bumper crop of coconuts, all of which had blown to the ground in the hurricane.

Having nothing better to do, the sailors decided to gather all the coconuts on the island into a single pile. They finished the job by nightfall, and then fell soundly asleep. But one sailor awoke in the middle of the night and became worried that he wouldn't receive his fair share of the milky palm fruit. He stole away to the pile and removed exactly his fair share of the coconuts, buried them, and went back to sleep. Another sailor awoke a short time later and did the same thing—buried his fair share of the coconuts and returned to sleep. Each of the three remaining sailors, in turn, did likewise.

The next morning, not suspecting what each of the others had done, the sailors divided the remaining coconuts, and each received his equal share. One coconut remained, which they gave to the monkey.

The Challenge: Write a program to determine the total number of coconuts in the original pile. There are multiple ways to solve the problem. Only a few lines of code are required in a short solution.

Let's first talk about the original problem, which is the classic problem in Diophantine analysis. We have these five sailors, see, and a big stack of coconuts. Each of them in turn divides the stack into five parts, but the division has one left over, which goes to the monkey. Each sailor then buries his fifth. When they meet in the morning to make the sixth division, it comes out exact.

The problem then is: what is the smallest number of coconuts in the original stack?

That was the problem that Ben Ames Williams used as the nub of his story "Coconuts" in the September 19, 1926 Saturday Evening Post. The story is reprinted in Clifton Fadiman's The Mathematical Magpie (Simon and Schuster, 1962). The problem entered the computing literature in Introduction to Electronic Computers by Dan McCracken and me (Wiley, 1961). A complete discussion of the Diophantine analysis involved in solving the problem mathematically is found in Martin Gardner's Second Scientific American Book of Mathematical Puzzles and Diversions (Simon and Schuster, 1961). However, the problem can be solved with a little common sense, a few simple equations, and some fifth grade arithmetic.

According to the terms of the problem, the final stack to be divided must be a multiple of 5. Thus, the final pile must be in this sequence:

5, 10, 15, 20, 25, 30, 35, 40, 45,... D = 5

Call these numbers N; N possible, to be more precise. (The D at the right is the common difference in the progression.)

Now, the stack that the last sailor had to deal with (call it M) must be given by:

$$M = (5/4)\cdot N + 1$$

Therefore the only useful values of N are those that are multiples of 4 (or else the calculation of the M values would lead to fractions). We have, then, (using successive letters for each stage, leading to X for the original stack):

N (useful):	20, 40, 60, 80, 100,...	D = 20
M (possible):	26, 51, 76, 101, 126,...	D = 25
M (useful):	76, 176, 276, 376, 476,...	D = 100
L (possible):	96, 221, 346, 461, 596,...	D = 125
L (useful):	96, 596, 1096, 1596,...	D = 500
K (possible):	121, 746, 1371, 1996,...	D = 625
K (useful):	1996, 4496, 6996, 9496,...	D = 2500

J (possible): 2496, 5621, 8746, 11871,...   D = 3125

J (useful): 2496, 14996, 27496, 39996,...   D = 12500

X (possible): 3121, 18746, 34371, 49996,...   D = 15625

so that the original stack (X) can be as small as 3121. To check this result:

Original pile: 3121
Left after 1st sailor hides 624: 2496
Left after 2nd sailor hides 499: 1996
Left after 3rd sailor hides 399: 1596
Left after 4th sailor hides 319: 1276
Left after 5th sailor hides 255: 1020
--and they each get 204 more coconuts the next day.

The problem submitted by Judy Lawlor is much simpler. We need a number, X, such that we could take 4/5 of it five times and then have a number of the form 5Q+1. That is:

$$.32768\,X = 5Q + 1$$

That fraction on the left reduces to:

$$\frac{4096}{12500}\,X = 5Q + 1$$

and the smallest value of X that fits is 12500, which is the desired result.

The box on the next page contains a program in BASIC to solve the <u>original</u> problem. To solve the problem given in <u>Small Systems World</u>, make these changes:

```
210 N = 5*Q + 1
300 M = 1.25*N
400 L = 1.25*M
500 K = 1.25*L
600 J = 1.25*K
700 X = 1.25*J
```

and the output will be:

```
12500, 28125,...                    D = 15625
```

```
100 Q = 0

200 Q = Q + 1
210 N = 5*Q

300 M = 1.25*N + 1
310 IF (M-INT(M)) > 0 THEN 200

400 L = 1.25*M + 1
410 IF (L-INT(L)) > 0 THEN 200

500 K = 1.25*L + 1
510 IF (K-INT(K)) > 0 THEN 200

600 J = 1.25*K + 1
610 IF (J-INT(J)) > 0 THEN 200

700 X = 1.25*J + 1
710 IF (X-INT(X)) > 0 THEN 200

800 PRINT X
810 GØTØ 200
```

and the output will be:

```
3121, 18746, 34371,...                 D = 15625
```

A word might be added about the method of solving such problems. We have a set of equations:

$$N = 5Q$$

$$M = (5/4)\cdot N + 1$$

$$\vdots$$

$$X = (5/4)\cdot J + 1$$

and if we substitute one equation into the next, to eliminate most of the variables, we get to:

$$X = \frac{15625}{1024} Q + \frac{8404}{1024}$$

That is, we have one equation in two variables (which is the characteristic of Diophantine problems), but with the added constraint that both X and Q must be integers. Therefore, we can rewrite the equation as:

$$X = \boxed{15Q + 8} + \frac{265}{1024} Q + \frac{212}{1024}$$

The boxed portion is an integer if Q is an integer. The other two terms must be made integral by the proper choice of Q, and that choice may not be clear at this stage. So we go on.

$$\frac{265}{1024} Q + \frac{212}{1024} = A \quad \text{(where A must be an integer)}$$

and, solving for Q:

$$Q = \frac{1024}{265} A - \frac{212}{265}$$

$$Q = 3A + \frac{229}{265} A - \frac{212}{265}$$

and the same argument applies--but notice that the numbers are now smaller. This reduction process continues:

$$A = B + \frac{36}{229} B + \frac{212}{229}$$

$$B = 6C - 5 + \frac{13}{36} C - \frac{32}{36}$$

$$\cdot\cdot \quad \cdot\cdot\cdot\cdot\cdot\cdot$$

$$E = \frac{10}{3} F + 2$$

until a level is reached at which an integral solution is evident. In this case, F = 0 will do nicely, making E = 2. Working back through the equations, we arrive at X = 3121, the smallest solution. If we let F = 3, we will get X = 18746, and so on. □

Personal Computing At The Threshold

by Fred Gruenberger

The material in this article is based on the keynote speech by Professor Gruenberger at the Personal Computer Festival held in conjunction with the National Computer Conference in May at Anaheim, California. With Mr. Gruenberger was a panel consisting of:

Bob Bemer, Honeywell
Reid Anderson, Verbatim
John Brackett, Softech
Werner Frank, Informatics

I have already spent a day exploring the exhibits here at the NCC and I am happy to report that today's children have their manners. Several small boys, perhaps 24 or 25 years old, generously offered to explain to me how computers work. One of them indicated that he could reveal to me how they <u>really</u> work, which is even better.

I'd like to talk about personal computing in terms of past, present, and future. Considering that it is just over five years now since the first Altairs caused the start of the shock wave, an overview of the field might be most appropriate.

The Personal Computing convention of 1978

It is only fair that you be given a chance to calibrate me. In November of 1978 there was a personal computing convention in Los Angeles and it might be revealing to review some of the pontifical observations I made in a talk at that affair. For example, there were at that time some 15 monthly publications devoted to this sub-field, and I observed that there certainly wasn't enough material each month to fill 15 magazines; hence, some of them would surely die. Today, by my count, there are 32 monthly publications, including a dozen or so just for the Radio Shack computer. Some publications have died, to be sure, but many new ones have sprung up. There seems to be something akin to infant mortality operating here: if a new magazine can survive to its fourth issue, it will probably make it.

Other items that seemed clear to me in 1978 were these: the industry was about to shift to 16-bit microprocessors; the use of Teletypes as output devices would attenuate rapidly; and paper tape as a storage medium was dead. My crystal ball was a bit cloudy, it would seem.

It was clear, and it is even clearer today, that the soldering iron phase of personal computing was over. The kit-building portion of our field, which had its greatest impetus from the ham radio people, had run its course. Unfortunately, the editors of our leading journals all belong to that group and can't wean themselves from it. Recent journals have had as much as 3/4 of their articles involving circuits and soldering. I don't believe that that is the thrust of personal computing.

Program pirating

One of the concepts that I would have thought would become clear was that quality software is expensive, and somebody must pay for it. The purveyors of personal computing software express shock and dismay at discovering that tapes can be copied and that they will be copied--wholesale. It is interesting to speculate as to why this disease--pirating--has affected only the personal computing world, and is of little concern in the big machine world (BMW). There are several reasons for the difference:

1. Vital programs (operating systems, language translators, utility programs, etc.) in the BMW are invariably bundled; that is, they are supplied "free" by the machine vendor, and thus the cost is evenly spread among all the users. This fact is partially true in the personal computing world, too. Every machine includes a BASIC interpreter of sorts; if you want a good one, or a compiler rather than an interpreter, you pay extra.

2. Software that is sold in the BMW is not trivial in either price or function. In the BMW there are no programs selling for $19.95; who would even set up a billing system at that price level? And in the BMW, no one buys a program to play complex versions of Star Trek.

3. Programs in the BMW are marketed exactly like hardware, and the price includes maintenance. A pirate will receive no maintenance. There are also licensing agreements and non-disclosure contract provisions in the BMW, which are not feasible at the $19.95 level.

4. In the BMW, without exception, it is a company that leases or owns the big machine, not an individual. Companies have more to lose by immoral or unethical conduct.

The problems of program protection, pirating, program maintenance--that is, all the things that are in any way involved with quality software--are the biggest problems that we have today, and it is going to take a long time to solve such problems. Pious editorials about the unfairness of program copying will not do it.

Speaking of pirating, there is another problem we have these days, and this one is common to all levels of the computing world. Because many of our problems--like program stealing--lead people to the courts, we have judges deciding highly technical questions and thus establishing precedents that we may not like. For example, we had a judge in Minneapolis decide that an obscure professor in Iowa, who had built what was essentially a binary adder back in the 30's, was the real inventor of the computer. Because of that bit of legal wisdom, that "fact" about the origins of the computer now appears in dozens of textbooks. More recently, in the CompuChess case, a judge declared that object code in a ROM was not protected by any laws. [Note: part of that case hinged on the lack of a copyright notice, which is the sort of point that judges can deal with, but the net result of the case still set a precedent about what could be protected.]

Our serial publications

Would you like to know how to make a lot of money fast? Take any article from the Communications of the ACM or from a Datamation of the 1950's (or, for numerical work, any text of the 1930's), and rewrite it in terms of BASIC, preferably botched so it won't even execute properly. Write at about the 5th grade level, and be sure to begin with "I have had absolutely no experience with computers up until I bought my KIM....which runs at near-infinite speed..." Send your new article to any of the five personal computing journals with the largest circulation; they will print it without question (and certainly without testing your program). Do not forget to state in the article that "...the program was written for the BASIC on my machine but with the simplest of changes it will run in any other BASIC." With diligence, you ought to be able to write three or four of these a day--clearly, someone is doing just that. If the article is really trite (like showing how to plot sine waves on a character or line output device) you should be able to get it into two journals simultaneously.

I think that this is the second big problem of our field; namely, the sad status of our literature. Consider, for example, a recent article on a "new" square root algorithm. I know some 19 distinct algorithms for square root (we devoted most of issue number 20 of POPULAR COMPUTING to that subject). This "new" one was for square roots of integers in the range from zero to 32767 (for which there are only 182 possible results).

The program given in the article had some small defects (admitted by the young author), such as slightly wrong answers for perfect squares and the inability to handle zero as an argument. It also closely resembled one of the oldest known algorithms--but it was labelled "new" and it got printed.

The sad fact is that the editors of our leading journals are not computer people at all. They are ham radio enthusiasts, or journalists, or salesmen. Last year, one of the journals published an article that had been copied verbatim from a 1953 article. The editor defended this blunder with the reasoning that the author had probably made an honest mistake, not knowing that the material had been previously published. The mistake is by the editor, who should know something about computing. Perhaps it's too much to ask for. Those editors are obviously skilled at turning out issues every month with little material coming in.

Standard languages

The personal computing industry has zeroed in on (interpretive) BASIC as its standard language. Unfortunately, there is no standard BASIC; we seem to have dozens of them, bearing only the faintest resemblence to each other. The popular Apple II comes with two of them, both badly produced, and gloriously incompatible with each other.

We are now seeing an effort to go to PASCAL, which may or may not cure this particular problem. One problem it will not cure is inherent stupidity in problem solutions; that is, the language itself will not do much to turn nonsense into wisdom, despite the claims of its fans. And, with all respect to Mr. Brackett (his company, Softech, markets UCSD PASCAL), I don't want to embrace any language that forces me to write "colon equal" for what could just as easily be simply "equal."

Now, a remark like that gets PASCAL devotees livid. Well, let's examine it. The argument is that "equals" has various meanings, and so it does. So how does Fortran distinguish between these meanings? The Fortran:

```
A = B
```

is really a left-pointing arrow, meaning that A is replaced by B. The algebraic equals is handled with:

```
IF (A.EQ.B) GØTØ 7
```

If it is necessary for the user to direct the compiler's attention to this distinction (and, they point out, "colon equal" sort of resembles a left-pointing arrow), then let's do it in the oddball case, not in the replacement case that is used in nearly every statement. It is the lack of human engineering that I am objecting to. I don't see why I have to do the compiler's work for it, just because the compiler writer is lazy. I like systems that work <u>for</u> me, so I can focus attention on my problem.

Incidentally, the PASCAL of 1986 will have at least 9 significant digits of precision in its floating arithmetic, and double precision capability for all arithmetic, logarithmic, and trigonometric operations. The current limitation of 6 digits is not only pathetic, but quite unrealistic. (Again, we devoted most of issue 52 of POPULAR COMPUTING to a discussion of the need for high precision arithmetic.)

Teaching the kids

There is currently a new wave of the fad of teaching computing to very young people. Each of the persons who lead such an experiment describes it at length in the literature and professes amazement at the wonderful results; namely, that the kids, from age 10 on up, can actually program in BASIC better than their teachers. The whole idea is hardly new, except that today it is easier to do, inasmuch as the personal machines are cheap. But the idea goes back to around 1959, when Richard Andree and George Heller and I began the first such experiments. I cannot speak for the other two men, but for myself, I have become convinced that it is a mistake to try to teach computing at too early an age. It seems to be directly analogous to the experiments in teaching calculus in high schools; it has been tried hundreds of times, and it is usually abandoned eventually, for the simple reason that it is a mistake. Very young people can memorize the rules of calculus, but the <u>understanding</u> of it takes maturity, and that can't be forced.

So it is, I believe, with computing. If it is pushed on to young people, they will surely learn the mechanics very quickly, and they will grind out trivial programs by the ton, but they will, in general, not learn computing. What is worse, having been exposed to the subject, they will very likely never learn computing. We are now getting in our universities the first wave of children who have already been exposed en masse to computers, either in well-meant high school classes, or through personal machines of friends, or parents, or at the nearby computer store. They are absolutely unteachable. They let you know that they are already experts (their third Tic-Tac-Toe program never loses) and they have no intention of sitting still for formal training. Why they take computing courses is a mystery.

I would like to point out to the industry that the B in BASIC stands for "Beginner's." BASIC is a fine language (well, one of them is, anyway) for learning the rudiments of programming. When strenuous efforts are made to twist it into an applications language (or, worse, a systems language), then it is time to call halt. If a tool is not right for a task, one should get the right tool, not try to sharpen up the wrong tool. The efforts to make BASIC fit advanced problems is part of what leads to the flood of incompatible BASICs. The personal computing field is not served by such misguided efforts.

The big surprise, to me, of the personal computing revolution is the extent to which professional computer people (including me) have embraced it. I find it difficult to explain why it is so fascinating to own a machine. There is virtually nothing I can do with it that I couldn't do as easily (and faster, and cheaper) on my employer's machine.

It is estimated (with the same one-digit precision with which we count everything in computing) that by the end of 1980 there will be half a million personal computers in use in this country. By 1986* we may have 3,000,000 machines of roughly the type we call personal machines today (but it is reasonable to speculate that they will be much different and certainly much more powerful). The impact of this infusion of computing power is difficult to predict.

**That is, six years from now. Predictions for six years into the future are most likely to avoid the pitfalls of over-optimism or over-pessimism.*

The grand piano analogy

Norman Sanders has suggested an analogy. What would be the effect of putting half a million grand pianos into homes across the country? In perhaps 498,000 homes, people would be playing Chopsticks on a grand piano. But in perhaps 2000 homes, the acquisition of a fine instrument might inspire someone to study sonatas and concertos who would not otherwise do so.

It is fun to try to peer into the future, say to 1986. Those three million machines, most of them with 16-bit words, will each have a minimum of 128K words of fast storage. The 75 journals of that time will be full of articles by the old-timers, recalling the Apple II and the TRS-80. The machines will have extensive built-in firmware, including an operating system, at least one high-level language (probably PASCAL) and a symbolic assembler, menu-driven diagnostic programs and numerous debugging aids. All of the built-in software will be reasonably idiot-proof and crash-proof, and will contain clearly labelled entry points for user alteration. All of this, being bundled, will be quality software and will be fully paid for by the users. The only software that will still be sold will be large applications packages (e.g., sophisticated word processors, business data processing packages, and esoteric languages) and these will be marketed in the same way that large packages are being marketed today. Machine troubles will be diagnosed by telephone hookup to the factory, and most repair work will be done by the user by exchanging modules, boards, and chips.

The normal speed of execution of personal machines will be 4,000,000 instructions per second; the top of the line machines will be four times as fast as that. The addressing schemes will span an address space of four billion words, with a maximum of 512K words in real storage and the balance in virtual storage, on disks.

The spread of private machines and data banks in industry and government will be a national problem of some significance, and legislation to control the machines and programmers will be the rule, rather than the exception. Between now and 1986 there will be two very large thefts of funds (one from banks and one from the government) involving small computers.

Some child of age 3 will emerge capable of writing small programs that work, and some child of age 16 will develop another "new" square root algorithm. And some editor will print articles about both of these.

Chapter 2 enumerates 96 specific *activities* frequently associated with the fifteen tasks discussed in Chapter 1. Milestone reporting, hands-on training, analysis of improvements, and the other activities can be placed in a matrix which shows the relationship of task/activity combinations to the five project phases. Customize such a matrix and you have a useful checklist for your particular development environment.

Chapters 3 through 7 address, in order, the five development phases. As each phase is discussed, specific design hints appropriate to the task/activity combination related to that phase are detailed. A total of 186 practical hints are distributed over the fifteen tasks as follows: design (61), project management (45) systems analysis (22), feasibility study (15), test development (10), business system requirements (7), documentation (7), and 1-4 hints on each of the remaining tasks (programming, training, conversion, physical facilities, installation, operation, system maintenance, and operations management).

Ranging in length from a single sentence to several pages, the 186 design hints yield numerous one-liners of wisdom:

Living with errors — At the risk of overstating the obvious, an on-line system should never abort.

Human factors hints — In prompted interactive dialogs, provide a fast path for highly skilled terminal operators.

Project staffing — If some critical task is at the fringes of the art and requires top talent, do not try to make do with more personnel having lesser skills.

Conversion strategy — Depending on the company and the current situation, getting a few hundred square feet near stable power in a benign environment, with good access to overhead or underfloor cable runs, may be a time-consuming political chore.

Training features — Some systems have provided special commands to allow a second terminal to be slaved to the first terminal so that a coach may follow the interactive dialog of his pupil without disturbing the training session.

Problem determination — System failures come in two varieties: hard and intermittent. Janitors running floor polishers, coffee pots in computer rooms (not recommended), maintenance of building air conditioning, and electric power fluctuations all have caused their share of intermittent problems.

Investment profiles — It is axiomatic that the project will have to be done over if it is not done right the first time.

The modest bibliography lists only 30 rather conservative and nontechnical references. At under $20, the cost of this attractively printed handbook is also modest. This reviewer could have prevented at least 20 headaches a year ago by simply reading these two simple hints about printers:

1. Operating systems software should provide service priorities. Applications should be designed to place production keyboarding at the highest priority, with other man-machine dialogs next, telecommunications next to that, and electromechanical devices (such as printers) last. Then deterioration in print performance is an indication that saturation is occurring.

2. On long reports, keep a short push-down list of page breaks so printing can be restarted at the beginning of the sheet in the event of printer malfunction/operator inattention.

In summary, here is a unique handbook of design ideas, independent of any one vendor's machinery or software. It enumerates both human factors and operational aspects the practicing designer must consider when preparing a first application for the small machine environment. Because on-line distributed systems fail publicly and impact their users immediately, there are some sharp and significant differences between what you can learn from this splendid book and what you may know from your experience with batch systems, which fail privately. Unless the design is carefully constructed, application programs for distributed processors or distributed terminals will fail hard, be difficult to restart, and will be unacceptable to their population of users. The author's concluding paragraph is timely and true:

"Remember that design is an intellectual, iterative process. Good design is a creative accomplishment. Designs which work, retain flexibility for change in the right places, and are in harmony with their users and their environment are the result of hard-working, well-seasoned analysts. Less experienced analysts can progress more rapidly if they exchange hints like those contained in this handbook."

Reviewed by Professor Jack Alanen
California State University, Northridge

Reprinted, with permission, from

DATA PROCESSING DIGEST 7/80

DATA PROCESSING DIGEST is published each month by Data Processing Digest, Inc., 6820 La Tijera Blvd., Los Angeles, CA 90045, USA. Telephone: 213/776-4334. Publisher: M. Milligan. Subscription Manager: Gisela Wermke.

BOOK REVIEW

Database: A Professional's Primer

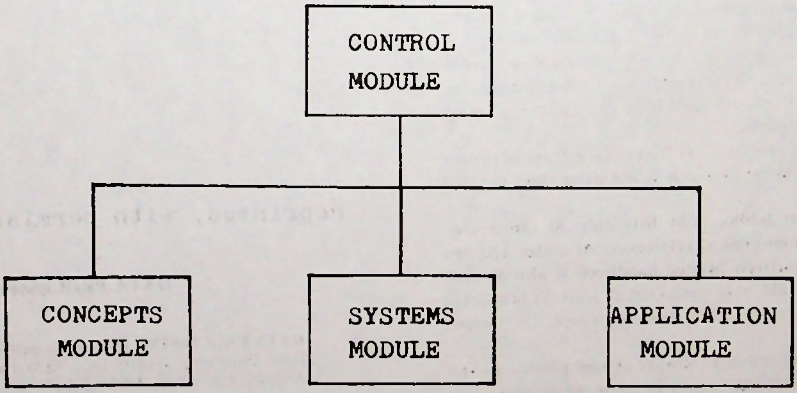
by David Kroenke
Science Research Associates, 1978, 323 pages, $20.

The "professional" referred to in the title of this book is the computer professional who has some knowledge of direct access storage devices, file access methods, and file management systems, but who has not yet been initiated into the arcane world of data base. Examples are:

- The programmer working on data base applications.
- The systems analyst who has just been promoted to data base administrator.

Newcomers to computers will find the book somewhat intimidating, and data base experts will find it somewhat superficial (and occasionally erroneous).

In an effort to pique the interest of the jaded programmer, the author characterizes the book as a "direct access device," and describes the organization of it by means of the following "volume table of contents":



The Control Module is intended for readers who want a quick overview of the subject and who can't spend more than an hour or two on the entire book. It provides a general discussion of data base (e.g., what are the advantages of data base processing?), followed by condensed versions of the three remaining modules. Readers who have wondered whether "data base" should be written as one word or two will be interested in the following explanation which appears in the Control Module:

- "Database" characterizes a system which
  - ▲ Integrates more than one logical record;
  - ▲ Represents more than one key per record;
  - ▲ Represents relationships among database records;
  - ▲ Provides program/data independence.
- "Data base" characterizes a system which does not meet these criteria, or which refers to that general area of computer activity for which database systems are intended.

The criteria which distinguish database systems from systems pretending to be database systems--and which are explained more fully later in the book--are technically valid; but to refer to the pretenders as "data base" systems is to subject those two little words to severe semantic overloading.

The Concepts Module provides the basic concepts necessary to understand generalized database management systems and their application. Concepts include:

- ▲ Data structuring concepts, such as linked lists, trees, and networks.
- ▲ Indexing concepts, such as primary and secondary keys.
- ▲ The concepts of a data model; that is, a class of data structures together with the language used to define and manipulate these structures.

Two specific models are described: the CODASYL Data Base Task Group (DBTG) model and the relational model introduced by E F. Codd. The exposition of the latter is done with unusual clarity and skill. The author is clearly enthusiastic about the relational model, to the detriment of an otherwise evenhanded style. For example, one advantage cited for the relational model is that it "tends to represent data as it exists." It turns out that "as it exists" refers to the author's graphical representation of data, in which entities are represented by nodes, and relationships between entities are represented by directed line segments connecting the nodes. Even then, the relational model provides a direct representation only for one-many relationships, with indirect methods being required in the many-many case. The essential equivalence of different models in terms of their information bearing capacity is made abundantly clear by the book's copious examples, but this key unifying concept is not discussed.

The Systems Module is devoted to a discussion of five commercial generalized database management systems (GBDMS). The systems are classified as hierarchic or network, according to the type of data model they use. The hierarchic systems are System 2000 (MRI Systems Corporation) and IMS (IBM). Since the hierarchic model is not described in the Concepts Module, its nature must be inferred from two specific implementations, which are quite different. The network systems are ADABAS (Software AG of North America), IDMS (Cullinane Corp.), and TOTAL (CINCOM). Of the three, only IDMS has a data model that is adequately explained in the Concepts Module (it is essentially the DBTG model). The data models of the other two can be inferred from the descriptions given, but are sufficiently different from one another and from the DBTG model that one wonders why they have been lumped under the single heading of network systems. In fact, remarks such as "The System 2000 hierarchy can be processed as a network" makes one wonder whether the author really understands the difference between hierarchies and networks.

The descriptions are necessarily superficial (for a book of this size) and occasionally misleading. For example, the logical data base construct in IMS is not adequately explained, thereby obscuring a salient characteristic of the system. The author doesn't understand how IMS uses the term "data base" (it is synonymous with "file"), resulting in incorrect statements of IMS capabilities. In an historical note, the evolution of the DBTG model from the IDS system is noted, but the terms "schema," "subschema," and "set" are incorrectly ascribed to IDS.

The use of "schema" (which has been used in mathematics for many years) to denote a data pattern or template was first proposed by the reviewer in 1969. The DBTG adopted the term to denote a data base pattern, and introduced the variation "subschema" to denote a subset of this pattern. DBTG also introduced the term "set" to denote an abstraction of the IDS chain construct.

Conspicuously absent from the Systems Module is any example of the relational model. While no relational system has yet achieved the commercial success of the five systems described, there are numerous implementations of the relational approach from which the author could have selected.

In the Applications Module, the author shifts from concepts and tools (data structures, GDBMS) to applications. The shift is confused by the continued use of "database system" to denote an information system; i. e., the procedures and programs developed by an enterprise (possibly with the aid of a GDBMS) to automate its information processing activities. A fairly detailed description is given of the information system development process: preliminary analysis, identifying objectives, specifying alternative information system designs (based on alternative GDBMS), evaluating alternatives (including the rating of GDBMS), and implementation. Noteworthy features are discussions of the use of rate of return as an evaluation criterion, and the placement of the data base administrator function with an enterprise.

This book is essentially a condensation of _Database Processing_: _Fundamentals_, _Modeling_, _Applications_ (SRA, 1977) by the same author. The latter is a textbook, aimed at a somewhat different audience. It contains all of the material in the present book, presented in a slightly different sequence. It contains some useful material not in the present book, including exercises at the end of each chapter, a description of a relational system, and an illuminating collection of data base case studies in which the experience of anonymous users with specifically-identified GDBMS is held up for public scrutiny. Much of the text and figures in the present book has been lifted bodily from the earlier book, producing anomalies of the type that can be expected from such a procedure: references to non-existent figures, figures which are not referred to, etc. If one is inclined to buy either of these books, the earlier one is probably the better buy. □

--reviewed by W. C. McGee

Y-Sequence

Suppose we define a sequence of numbers, Y, for which Y_1 is defined to be <u>one</u>, and subsequent terms are given by:

$$Y_n = \frac{2Y_{n-1} - n}{n} \qquad \text{for n odd}$$

$$Y_n = \frac{Y_{n-1} + n}{n} \qquad \text{for n even}$$

So that the first few terms are as follows:

1	1
2	1.5
3	0
4	1
5	- .6
6	.9
7	- .742857142
8	.907142857
9	- .798412698
10	.920158730
11	- .832698413
12	.930608465
13	- .856829467
20	.955003279
21	- .909047307
50	.980800028
51	- .961537254
100	.9902000008
101	- .980392079

The even-numbered terms appear to approach <u>one</u>, but very slowly; the odd-numbered terms approach minus one.

Here's the Problem: The even-numbered terms first exceed the value .99 at the 98th term. The value .999 is first exceeded at the 998th term. The value .9999 is first exceeded at the 9998th term. From this data, can we reach a conclusion as to when the value .99999 will first be exceeded by an even-numbered term? (The initial values for the first 4 terms are to be disregarded here.)